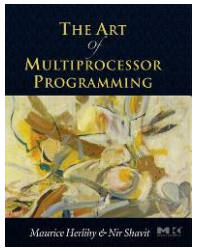
Real-Time Systems

Part 6: Concurrency

Contents

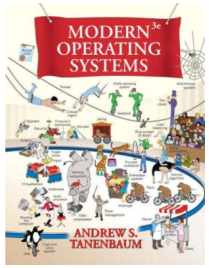
- Introduction
- Processes
- Threads
- Interrupts
- Concurrency problems and their solutions
- Inter-Process-Communication (IPC)

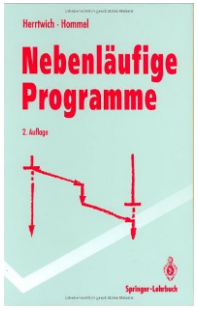
Literature



Maurice Herlihy, Nir Shavit, The Art of Multiprocessor Programming, 2008

A.S.Tanenbaum, Modern Operating Systems, 2008





R.G.Herrtwich, G.Hommel, Nebenläufige Programme 1998

- Edward Lee: The Problem with Threads: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf
- http://www.beyondlogic.org/interrupts/interupt.htm
- http://www.llnl.gov/computing/tutorials/pthreads/

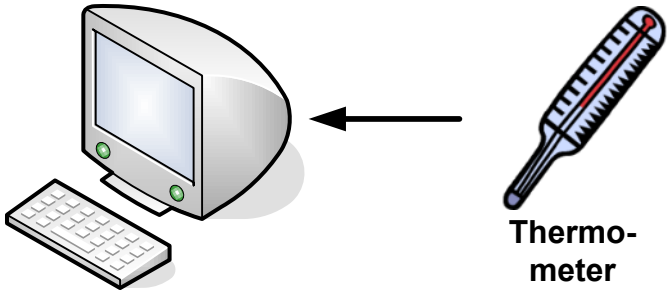
Definition Concurrency

- **Common meaning**: Concurrent events are not causally dependent on each other. Events (sequences of events) are concurrent, if none is the cause of another one.
- **Meaning in computer science**: Concurrency describes the property of code to be runnable in parallel, instead of being executed sequentially.
- Instructions can be run in parallel (pseudo parallel), if they are not dependent on the respective results
  - The parallel execution of several independent processes on one or multiple processors is called **multi-tasking**
  - The parallel execution of subsequences within a process is called **multi-threading**

Motivation

- Reasons for concurrent execution of programs in real-time systems:
  - Real-time systems are often distributed (Systems with multiple cpus).
  - Often real-time critical and non-real-time tasks are executed in parallel
  - In reactive systems the maximum response time is often limited
  - Mapping of the parallel sequences in the technical process
- **BUT:** often small (single-cpu) real-time systems do not perform parallel execution of code, because of the time-overhead involved in the process-management**.** Nevertheless pseudo-parallel execution also happens here in the main-program and the interrupt service routines.

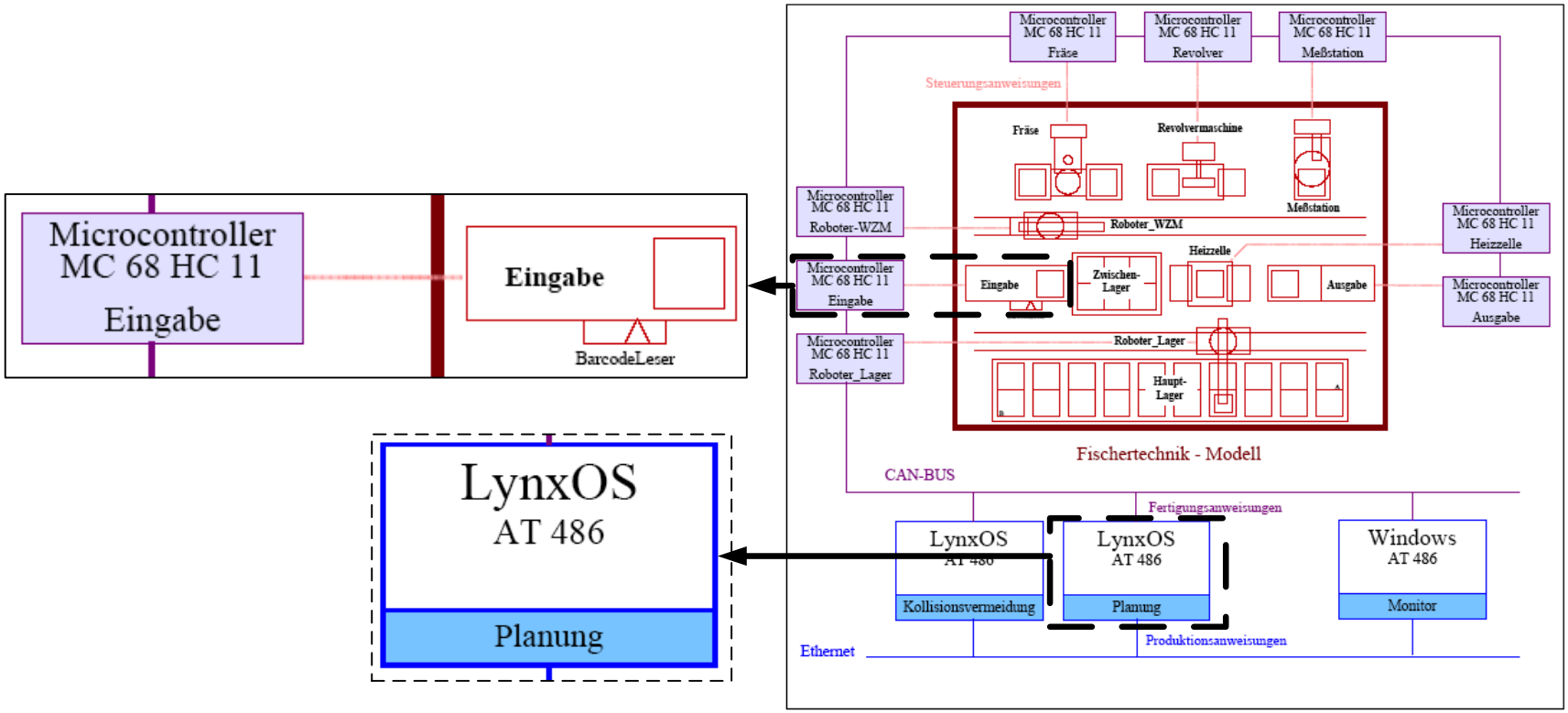
Use cases for concurrency: Interrupts



Signal if critical temperature level is reached
⇨**Interrupt**

Common use case: mainly for the interaction with external hardware

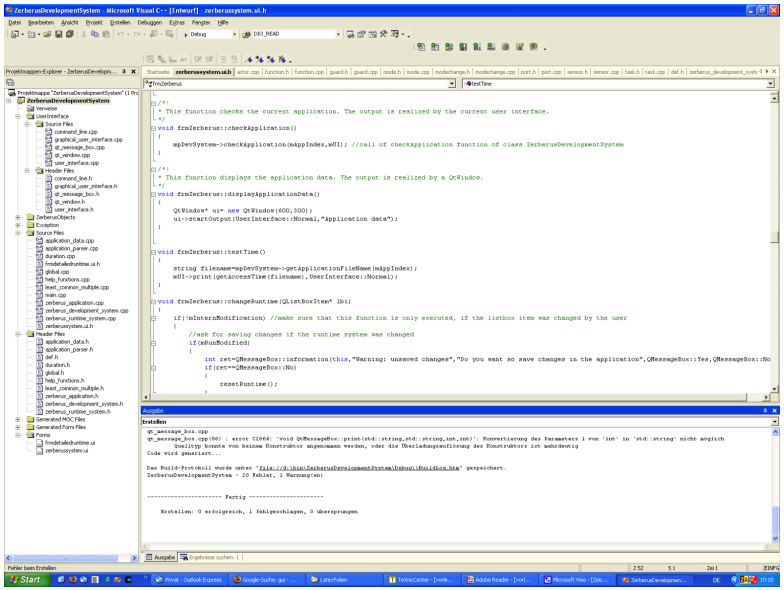
Use cases for concurrency: Processes



Distributed system for controlling a industrial plant ⇨ **prozesses (tasks)**

Common use case: distributed systems, different application on a single processor

Use cases for concurrency: Threads



Responsive to user input despite calculations (e.g. compiling a program)

⇨ **lightweight processes(Threads)**

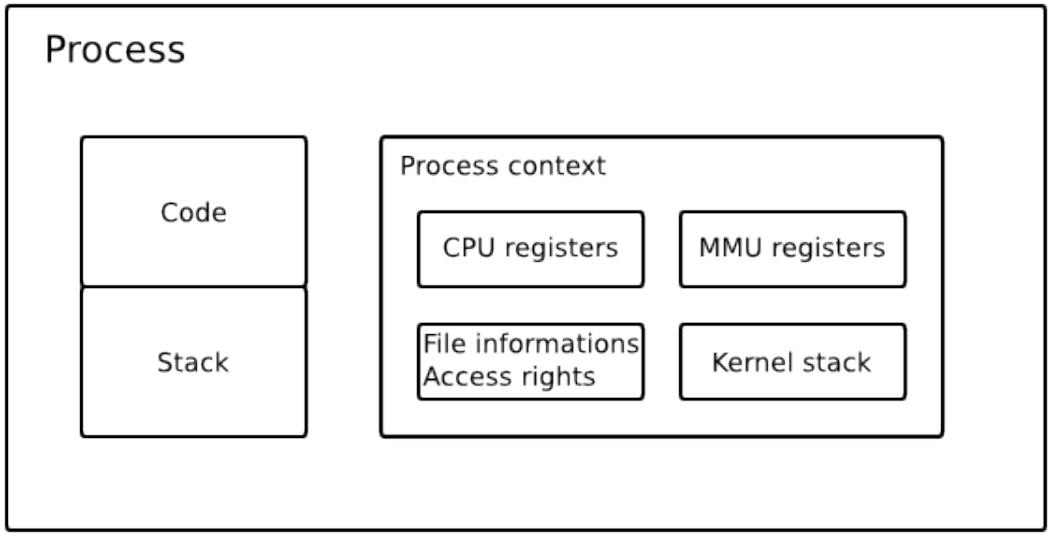
Common use case: different computations in the same application context

Concurrency

Processes

Definition

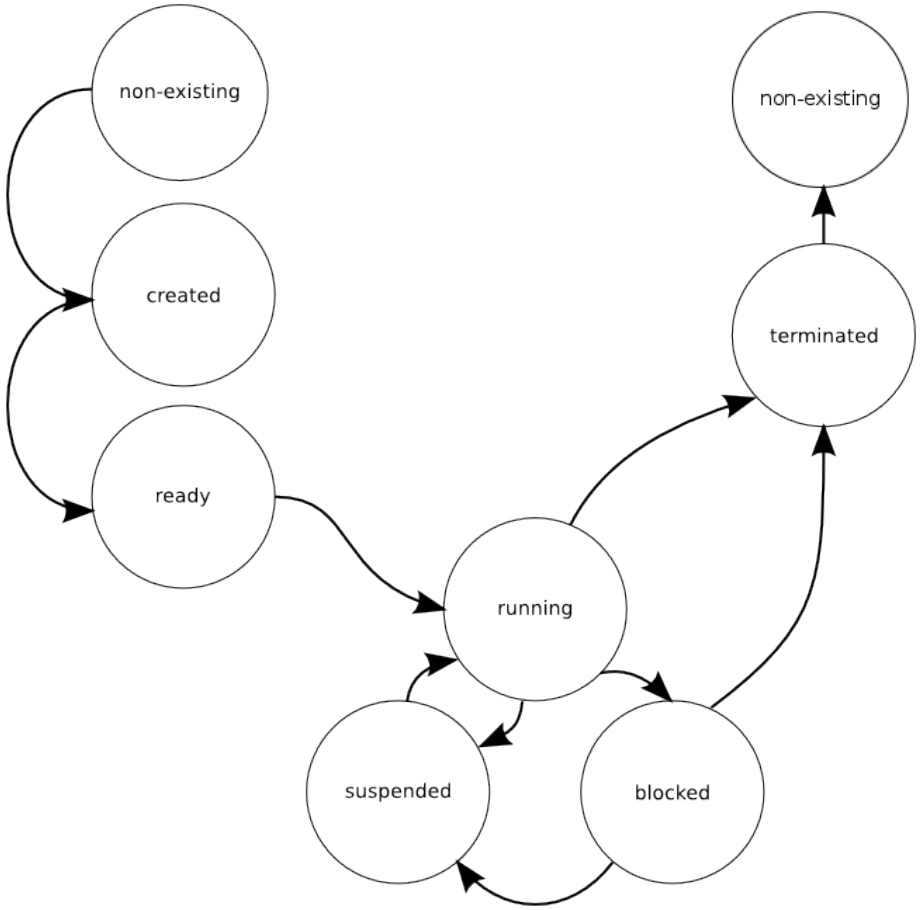
- **Process**: Abstraction of an program being executed
- The whole state information of the resources for a program is considered as one unit and called process
- Processes can spawn new processes ⇨ parent- and child-processes



Process execution

- Process execution requires certain resources:
  - CPU time
  - memory
  - Other resources (e.g. special hardware)
- Execution time depends on:
  - CPU performance
  - Availability of resources
  - Input parameters
  - Delay due to other/more important tasks

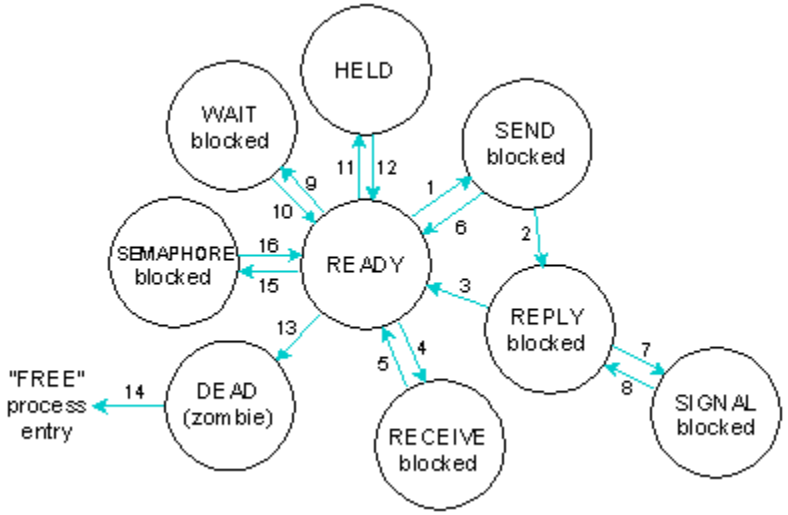
Process states (common)



Processes in QNX[1]

The transactions depicted in the previous diagram are as follows:

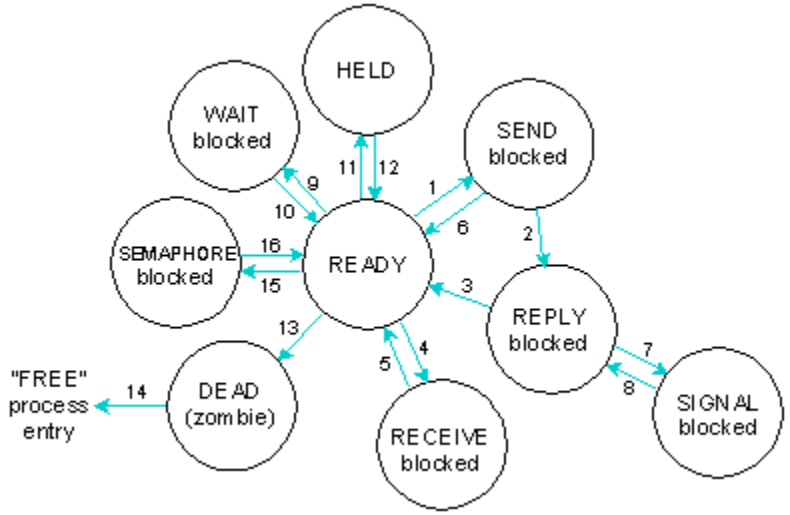
1. Process sends message.
2. Target process receives message.
3. Target process replies.
4. Process waits for message.
5. Process receives message.
6. Signal unblocks process.
7. Signal attempts to unblock process; target has requested message signal catching.
8. Target process receives signal message.



[1] http://www.qnx.com/developers/docs/qnx_4.25_docs/qnx4/sysarch/proc.html#LIFECYCLE

Processes in QNX

9. Process waits on death of child.
10. Child dies or signal unblocks process.
11. SIGSTOP set on process.
12. SIGCONT set on process.
13. Process dies.
14. Parent waits on death, terminates itself or has already terminated.
15. Process calls *sem_wait()* on a non-positive semaphore.
16. Another process calls *sem_post()* or an unmasked signal is delivered.



Questions for real implementations

- Which resources are necessary?
- Which execution duration do single processes have?
- How can processes communicate?
- When should which process be scheduled?
- How can processes get synchronized?

Process classes

- periodic vs. aperiodic
- static vs. dynamic
- Importance of the processes (critical, necessary, not necessary)
- Memory resident vs. swappable
- Process can be executed
  - on a single machine (pseudo-parallel)
  - on a multi-core/-cpu system with shared memory
  - on a multi-cpu system with separated memory

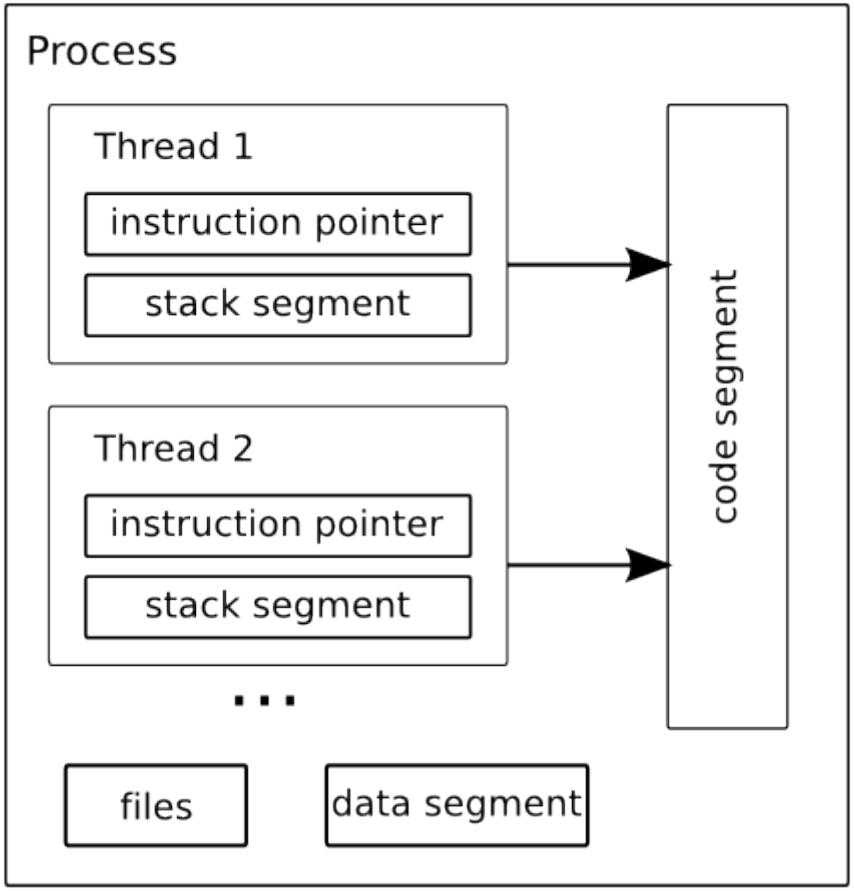
Concurrency

Threads

Lightweight processes (threads)

- Memory requirement of a single process is normally big (CPU-data, state information, details about files and devices...).
- For switching processes the corresponding data must be exchanged ⇨ high system load, time-consuming
- Many systems do not need complete new processes. Instead different execution paths are necessary, which operate on the same data⇨ Threads

Threads


Process
Thread 1
instruction pointer
stack segment
Thread 2
instruction pointer
stack segment
code segment
files
data segment

Processes vs. Threads

- Processes have different memory regions, threads share the same memory
- The management overhead of threads is much smaller
- Efficiency: for switching threads in the same program context it is not necessary to exchange the complete process information
- Threads can communicate using the common memory
- Access to the memory of other processes leads to errors
- Problems using threads: operating on the same data can lead to conflicts

Concurrency

Interrupts

Interaction of the computing system with the environment

- The system must be able to react to changes in the environment (e.g. a button has been pressed)
- **1. Polling**
  I/O register are permanently checked for changes and in the case of changes special response-routines are called
  - Advantages:
    - For a few I/O registers the latency is extremely small
    - In the case of very many events the timing behavior does not change drastically
    - Communication is synchronized with the program execution
  - Disadvantages:
    - Most I/O checks are unnecessary
    - High cpu-load
    - Response-time grows with the number of event-sources

Interaction of the computing system with the environment

- **2.Interrupts**
  the execution of current running program is stopped. Action is then taken according to the priority of the encountered event.
  - Advantage:
    - Resources are only used if necessary
  - Disadvantages:
    - Non-determinism: interrupts can happen asynchronous to the path of execution (and the process-state)

Interrupts

- **Interrupts**: execution of the current main program is halted and execution of a special interrupt service routine (ISR) is started. After completion of the ISR the execution of the main program is normally resumed at the previous location.
- **Synchronous interrupts**: occur always at the same code location. Also called traps, exceptions and “software-interrupts”.
- **Asynchronous interrupts**: time and location of occurrence are unknown. Asynchronous interrupts are often just called interrupts or hardware-interrupts, because they are generated by external. They build the bridge between the hard- and software.

Synchronous interrupts (traps/exceptions)

- Are triggered by the program itself. Therefore the same program executed with the same parameters will normally have the same interrupt at the same code location.
- **Triggered by errors** – exception, examples:
  - Arithmetic error (division by zero, overflow, not-a-number NaN, …)
  - Memory error (page Fault, segment fault, memory full, …)
  - Instruction error (illegal instruction, privileged instruction, bus error, …)
  - Peripheral error (end-of-file EOF, channel blocked, unknown device, …)
- Return to the location of occurrence only if the erroneous situation has been successfully cleared in the ISR otherwise the program is aborted (resumption vs. termination)
- **Triggered by specific instruction**: breakpoint, SWI, TRAP, INT, … either for the purpose of debugging or execution of system calls
  (e.g. MS-DOS „INT 21h“, http://en.wikipedia.org/wiki/MS-DOS_API)
- Traps can also be used to test a hardware-interrupt service routine

Asynchronous interrupts

- Triggered by external processes the time and code-location are normally not known also not reproducible.
- Examples:
  - Signaling of “normal” events by external peripherals (timer, switch, …)
  - Warning signals of hardware (low battery, „watchdog-timer“ timed out, …)
  - Completion of an I/O operation (complete word received/sent through a serial line, Coprocessor operation completed (DMA, FPU), …)
- Handling of interrupts does not have any side effects – therefore the context of the main program is not allowed to be modified after returning form the ISR.
- **BUT**: it is typical that the ISR and programs communicate using shared memory or global variables.
- Hardware interrupts have been treated in detail in chapter 3

Concurrency

Problems

Problems

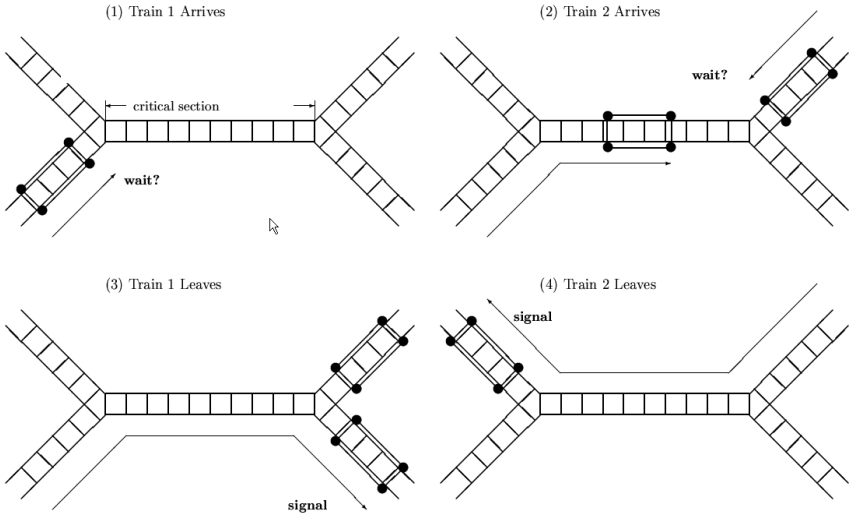
- **Race Conditions**:
  - Threads/processes are reading or writing shared data, which leads to results depending on the exact execution order of instructions
  - Solution: Introduction of **critical sections** and **mutual exclusion**.
- **Starvation**:
  - a process is perpetually denied necessary resources and therefore waiting forever. Important: reasonable implementation of waiting queues for resources e.g. priority-based queues
- **Priority inversion**:
  - High priority process can be delayed by low-priorityy processes, which have acquired resources. This problem will be treated in detail in the scheduling chapter

Necessary conditions for the mutual exclusion

- A good solution for the mutual exclusion fulfills at least 4 criteria:
  1. Only one process is allowed to enter the critical section
  2. No assumption can be made about the speed and the amount of processors
  3. No process is allowed to block another process outside the critical section
  4. Every process has only to wait a finite time to enter the critical section

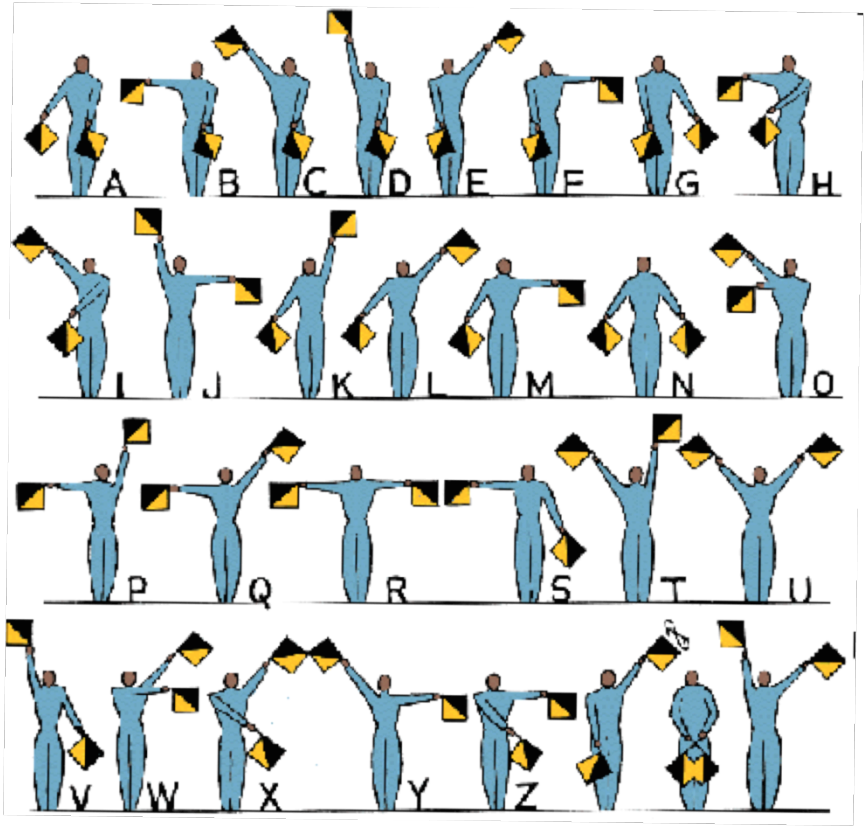
Critical sections

- To protect a critical section means have to be taken to stop processes/threads to enter the region simultaneously
  - If **only one** process/thread is allowed to enter the critical region – it is called mutual exclusion
  - If we want to avoid that several instances of different process-classes enter the critical section, then this problem is called the **reader-writer-problem** (e.g. several readers are allowed to enter, but a writer needs exclusive access).



Techniques for protecting critical sections

- Everyday life examples for protecting critical sections are:
  - *Railroad signals*
  - *Traffic lights at crossroads*
  - *Room-locks*
  - *Distribution of tickets*
- Method of choice for protecting critical sections in software: the semaphore, from ancient greek sign (sêma) and bearer (phoros)



Wrong solution: Using a global variable

Thread A

```
bool block = false;//global variable
...
while( block )  //busy waiting
    ;
block = true;
/* critical section  */
block = false;
...
```

Thread B

```
while( block )  // busy waiting
    ;
block = true;
/* critical section  */
block = false;
...
```

- This implementation is not correct:
  - the process could be directly interrupted after the while statement and possibly resumed after the block variable has been modified by the other process
  - Additionally the solution is inefficient (busy waiting)

1. Possibility: Peterson 1981 (for two processes/threads)

Thread A

```
int turn = 0;
bool ready[ 2 ];
ready[ 0 ] = false;
ready[ 1 ] = false;

...
ready[ 0 ] = true;
turn = 1;
while( ready[ 1 ] && turn == 1 )
    ; // busy waiting
/* critical section */
ready[ 0 ] = false;
...
```

Thread B

```
...
ready[ 1 ] = true;
turn = 0;
while( ready[ 0 ] && turn == 0 )
    ; // busy waiting
/* critical section */
ready[ 1 ] = false;
...
```

- Exclusion is granted, but „busy waiting“ wastes computational resources
- The extension to N processes is known as „Lamport‘s Bakery algorithm“

2. Possibility: Disabling interrupts

- Process switching is always caused by an interrupt (e.g. new event, timeout)
- Disabling interrupts prevents context switches and is a viable measure to protect critical sections
- Advantages
  - Simple to implement, no further concepts necessary
  - Fast execution (toggle bits in register)
- Disadvantages:
  - Not suited for multi-processor systems
  - No treatment of interrupts during execution of the critical section
  - Long locking critical for real-time systems

5.4.5 Interrupt Enable Clear register (VICIntEnClear - 0xFFFF F014)

This is a write only register. This register allows software to clear one or more bits in the Interrupt Enable register (see Section 5.4.4 "Interrupt Enable register (VICIntEnable - 0xFFFF F010)" on page 52), without having to first read it.

Table 42: Software Interrupt Clear register (VICIntEnClear - address 0xFFFF F014) bit allocation
Reset value: 0x0000 0000

Bit	31	30	29	28	27	26	25	24
Symbol	-	-	-	-	-	-	-	-
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	23	22	21	20	19	18	17	16
Symbol	-	-	AD1	BOD	I2C1	AD0	EINT3	EINT2
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	15	14	13	12	11	10	9	8
Symbol	EINT1	EINT0	RTC	PLL	SPI1/SSP	SPI0	I2C0	PWM0
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	7	6	5	4	3	2	1	0
Symbol	UART1	UART0	TIMER1	TIMER0	ARMCore1	ARMCore0	-	WDT
Access	WO	WO	WO	WO	WO	WO	WO	WO

3. Possibility: Semaphore

- Semaphore introduced by Edsger W. Dijkstra in 1965.
- The semaphore is a data-structure, consisting of a counter variable `s`, and the functions `down()` or `wait()` (resp. `P()`, von probeer te verlagen) and `up()` or `signal()` (resp. `V()`, von verhogen).

```
void Init( Semaphore s, int v )
{
    s = v;
}
```

```
void V( Semaphore s )
{
    s = s + 1;
}
```

```
void P( Semaphore s )
{
    while( s <= 0 )
        ;
    s = s - 1;
}
```

- Before entering the critical section a process needs to acquire the semaphore by calling the `down()` function. On leaving the section the semaphore is released using the `up()` function.
- **Important assumption**: the execution of the function `up` and `down` does not get interrupted and is executed atomically (**atomic execution**)
- As long as the critical section is occupied (`s <= 0`) the calling process is blocked

Example: bank account

- A Semaphore "semAccount" allows us to keep a bank account "account" consistent during a parallel write and read access from two processes

Thread A

```
P( semAccount );
x = readAccount( account );
x = x + 500;
writeAccount( account, x );
V( semAccount );
```

Thread B

```
P( semAccount );
x = readAccount( account );
x = x - 200;
writeAccount( account, x );
V( semAccount );
```

- For mutual exclusion a binary semaphore with the two states 0 (free) and 1 (occupied) is used. Binary semaphores are also called Mutex (mutual exclusion).

Extension: counting semaphore

- If the value of the semaphore is bigger than 1, then the semaphore is also called counting **semaphore.**
- Counting semaphore example: in case of the reader-writer problem the number of simultaneous readers can be limited to 100

```
semaphore sem_reader_count;
Init(sem_reader_count,100);
```

- Every reader-process executes the following code:

```
P(sem_reader_count);
read();
V(sem_reader_count);
```

- Reader-writer problems occur in various variants according to the priority of the processes (no priorities, reader-priority, writer-priority).

Implementing Semaphores

- Implementation of semaphores need special hardware support and instructions. The semaphore itself is a critical section and the functions `up()` and `down()` must be un-interruptible. Otherwise the semaphore can become inconsistent.
- Instructions which are not interruptible are called **atomic.**
- Different variants for the implementation:
  1. Short-term blocking of process switches during execution of the functions `up()` and `down()`. Realization by disabling interrupts.
  2. **Spinlock**: programming technique based on busy waiting. Indepedent of the OS and also applicable in multi cpu systems, but it comes along with a big waste of computational resource. In contrast the techniques of 1 and 2 can be implemented efficiently using message queues.
  3. **Test&set**-instruction: most processor offer a special „t**est&set**“ (or test&set lock) instruction. This instruction atomically loads the contents of a memory location in a register and also stores the modified register atomically to the memory location.

Implementing Semaphores

Test&set-instruction for multi-cpu systems

- Problem: simultaneous access to the same memory region from multiple cpus
- For the test&set instruction the exclusive access to the memory has to be granted
  ⇨Bus Locking
- Details can always be found in the manuals and specifications of the processors (e.g. Intel Architecture Software Developer's manual)

```
enter_region:     ; A "jump to" tag; function entry point.

tsl reg, flag             ; Test and Set Lock; flag is the
                          ; shared variable; it is copied
                          ; into the register reg and flag
                          ; then atomically set to 1.
cmp reg, #0               ; Was flag zero on entry?
jnz enter_region          ; Jump to enter_region if
                          ; reg is non-zero; i.e.,
                          ; flag was non-zero on entry.
ret                       ; Exit; i.e., flag was zero on
                          ; entry. If we get here, tsl
                          ; will have set it non-zero; thus,
                          ; we have claimed the resource as-
                          ; sociated with flag.
```

Improved concept: Monitors

- One drawback of semaphores is that the developers has to explicitly acquire und release the requested resources using down/up (P/V) pairs.
- If a corresponding up (V) statement is missing, then easily a deadlock is created. These kind of errors are typical very hard to find!
- For easier and also less error-prone treatment of critical regions the concept of monitors was introduced (Hoare 1974, Brinch Hansen 1975) :
  - A **monitor** is a unit of data and methods operating on the data, which can be accessed only by one process at a time
  - If more than one process wants to access the monitor then all processes except one are queued up and get blocked
  - If a process leaves the monitor, then one of the queued processes is dequeued and allowed to access the methods and data of the monitor
  - Signaling is done within the monitor and has not to be programmed by the user

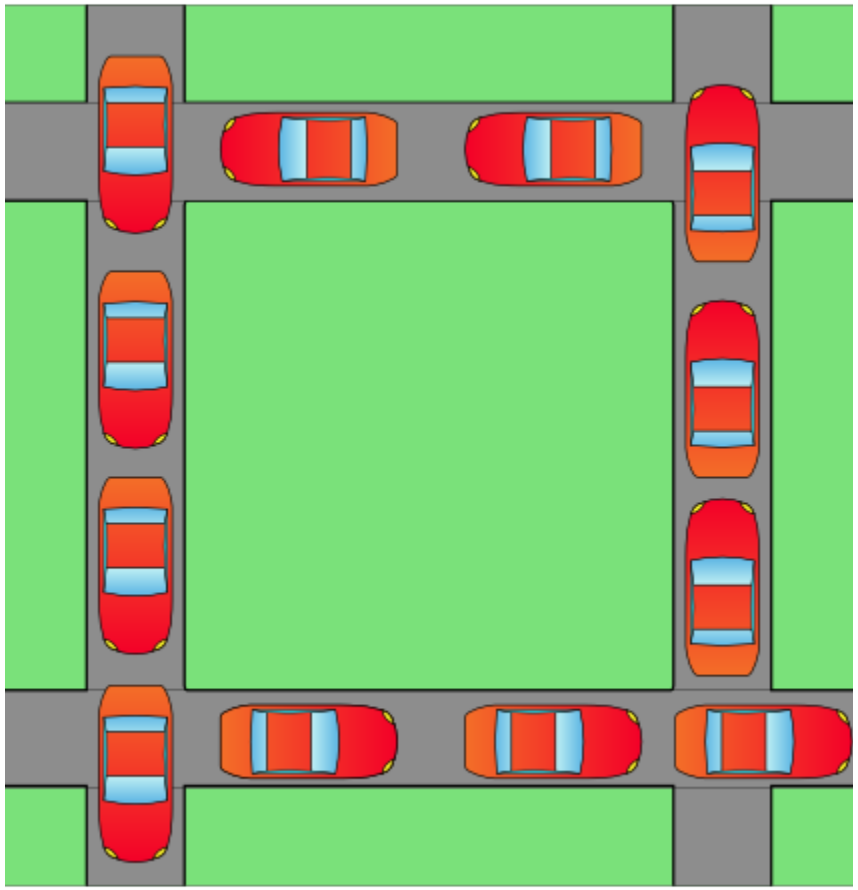
Example: Monitors in Java

- Java implements monitors using `synchronized`-methods. Only one process at a time is allowed to enter these methods.
- **Note**: normally higher-level construct like the monitor are implemented using semaphores (or realized using the even simpler TSL instructions).
- The monitor concept in Java can be used to implement semaphores like in the example on the right
- `wait()` and `notify()` are defined for each object in Java

```
public class Semaphore {
    private int value;

    public Semaphore( int initial ) {
        value = initial;
    }

    synchronized public void up() {
        value++;
        if( value == 1 )
            notify();
    }

    synchronized public void down() {
        while( value == 0 )
            wait();
        value--;
    }
}
```

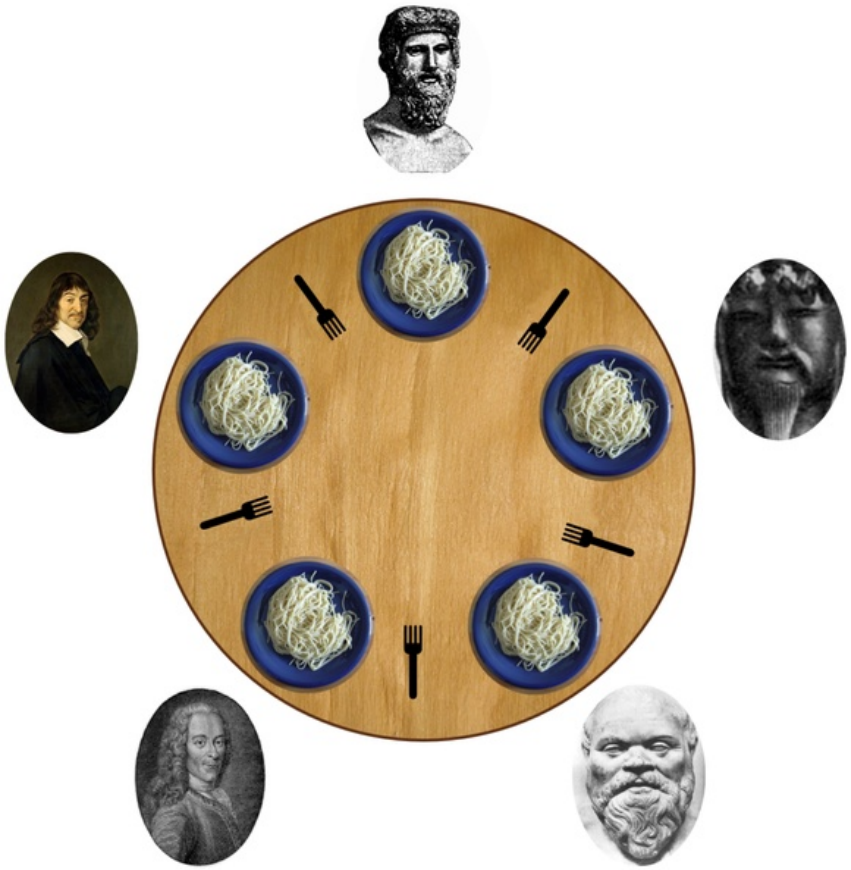
Deadlock conditions

- Even correct usage of semaphores ands monitors can lead to deadlocks
- A deadlock situation can arise if all of the following conditions hold simultaneously in a system (Coffman, Elphick und Shoshani 1971)
  1. **Mutual exclusion**: There exists a set of exclusive resources, which are either free or occupied by a single process.
  2. **Hold-and-wait**: A process is currently holding at least one resource and requesting additional resources held by another process.
  3. **No preemption**: Resources can not forcibly taken by another process or the OS once they have been acquired; they must be released voluntarily.
  4. **Circular wait**: There must exists a circular chain of processes, which wait for the resources currently belonging to the next process in the chain.
- Unfulfillment of any of these conditions is enough to preclude a deadlock from occurring.



Example: dining philosophers

- Classic example for deadlocks: "Dining Philosophers" (Dijkstra 1971, Hoare 1971)
- 5 philosophers (processes) sit around a table. In front of each philosopher is table with food. Each philosopher needs to forks (resources) for eating, but only 5 are available on the table.
- The philosophers are thinking and discussing. If one of them gets hungry, he takes the left forks and then the right one. If one of the forks is not available then he waits for it (without putting back the other one). After eating he puts the forks back.
- Problem: if all philosophers get hungry at the same time, then they pick up their left fork and therefore also the right of their neighbors. All philosophers will wait forever for the right forks (deadlock).
- If a philosopher does not put back his fork, then his neighbor starves **(starvation)**.



Inverse counting semaphore?

- Task: Implementation of the reader-writer problem with writer-priority.
- Explanation:
  - Data can either be accessed by many reader or one writer.
  - As soon as a writer wants to access the data no additional readers (or writers) should be allowed to access the data. Upon signaling of the writer-event all reading operations can be finished properly, afterwards the writer can perform its operations.
- Problem: Often it is tried to solve the problem using an “invers counting semaphore”, which is available if the counter is at 0 and block otherwise.
- What is the correct solution?

Concurrency

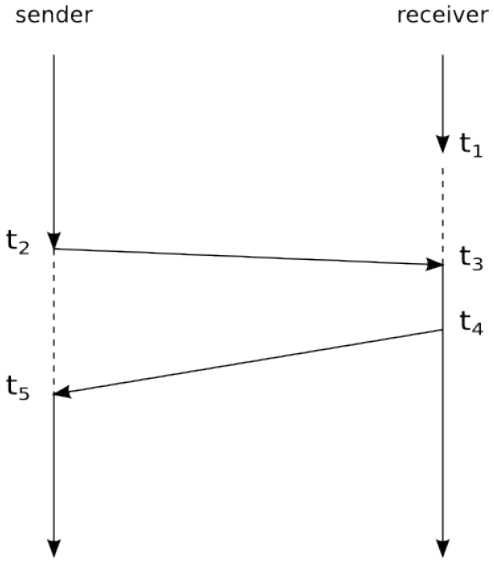
Inter-Process-Communication (IPC)

Inter-Process-Communication (IPC)

- Necessity of IPC
  - Process work in different process-spaces or even different processors.
  - Processes eventually need results from other processes
  - Implementing mutual exclusion makes use of signaling mechanisms
- Classification of communication
  - synchronous vs. asynchronous communication
  - Pure events vs. messages with values

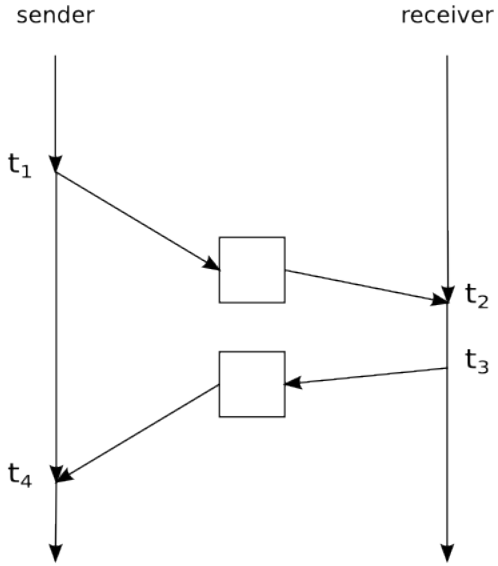
Synchronous vs. asynchronous

synchronous



- t1: receiver waits for message
- t2: sender sends message and blocks
- t3: receiver gets and processes message
- t4: processing finished and reply is send
- t5: sender receives message and continues

asynchronous



- t1: sender sends message to buffer and continues
- t2: receiver receives message
- t3: receiver writes result to buffer
- t4:sender reads result from buffer

(not shown: additional checking of buffer or waiting)

IPC-Mechanisms

- Data-streams:
  - Direct data exchange
  - Pipes
  - Message Queues
- Signaling events
  - Signals
  - Semaphores
- Synchronous communication
  - Barriers/Rendezvous

Concurrency

Inter-process-communication using data-streams

Direct communication

- Semaphores and monitors protecting data-structures are well suited for directly exchanging data
  - Fast communication, since the memory can be accessed directly.
- On the other side communication can only happen locally and also the processes need to be tied very tightly.
- Programming languages, operating systems and also middle-wares offer more comfortable methods for exchanging data.
- Basically the data exchange happens using the functions `send(receiver address, &message)` and `receive(sender address, &message)`.

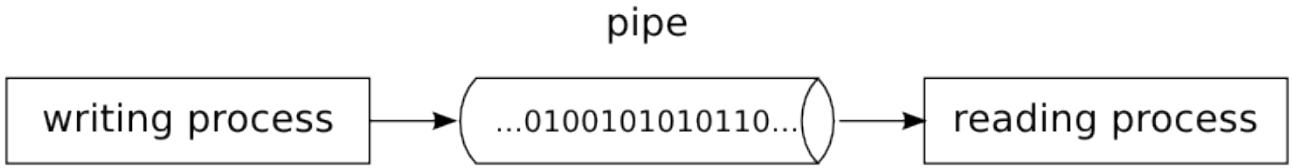
Questions regarding the data exchange

- Message-based or data-stream?
- Local or distributed communication?
- Parameters for the communication:
  - With/without acknowledgement
  - Message loss
  - Time intervals
  - Order of the messages
- Addressing
- Authentication
- Performance
- Security (encryption)

This chapter focuses on local communication. Other communication protocols and hardware have been treated in the communication chapter.

Pipes

- A pipe refers to a buffered, uni-directional data-connection between to processes according to the **First-In-First-Out-** (**FIFO**-) principle.
- Named pipes (similar to filenames) allow processes from different origins to read/write pipes.
- For processes form the same origin (e.g. parent-,child-process) anonymous pipes can be used.
- Communication is always asynchronous



POSIX Pipes

- POSIX (Portable Operating System Interface) tries to increase the portability of programs by standardizing the system calls between different operating systems.
- POSIX.1 defines the following functions for pipes:

```
int mkfifo(char *name, int mode ); /* creates a named pipe*/

int unlink( char *name );/* deletes a named pipe */

int open( char *name, int flags ); /* opens a named pipe */

int close( int fd ); /*  closes a pipe*/

int read( int fd, char *outbuf, unsigned bytes ); /*read data from a pipe*/

int write( int fd, char *outbuf, unsigned bytes ); /*write data to a pipe*/

int pipe( int fd[2] ); /*creates a unnamed pipe*/
```

Drawbacks of Pipes

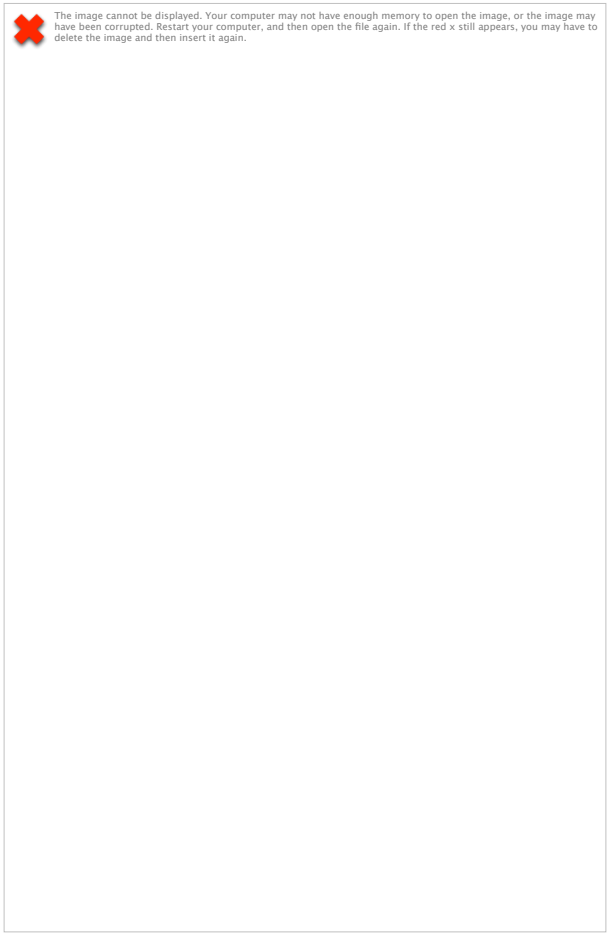
- Pipes are not message-oriented (data is not bundled into messages).
- Data can not have any priorities
- The memory needed for the pipes is allocated during runtime
- Implementation notes:
  - No data can be retained
  - The open function blocks until also the other end calls open (can be avoided using the O_NDELAY flag).
- Solution: message-queues

Message Queues

- Message queues are an extension to pipes.
  In the following we will talk about message queues as proposed in POSIX 1003.1b (POSIX real-time extension)
- POSIX Message Queue properties:
  - On message-queue creation the necessary amount of memory is reserved.
    ⇨Memory is not allocated during the first write access.
  - Communication is message-oriented. Die number of messages can therefore be queried.
  - Messages can have priorities ⇨ It is much easier to give time guarantees

Message Queues

- Write access in standard systems: the writing/ sending process is only blocked, if the memory of the data-structure is already full. **Alternative in real-time systems: error message without blocking**
- Read access in standard systems: the reading/ receiving process is blocked until a message arrives. **Alternative: error message without blocking**
- A illustrative example is a printer spooler: printing jobs are accepted and forwarded to the printer one after another.



POSIX Message Queues

- POSIX defines the following message queue functions:

```
/* create new message queue */
mqd_t mq_open(const char *name, int oflag, ...);

/* close message queue */
int mq_close(mqd_t mqdes);

/* removes message queue */
int mq_unlink(const char *name);

/* send message */
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio)
/* receive message */
size_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio);

/* set attributes */
int mq_setattr(mqd_t mqdes, const struct mq_attr *mqstat, struct mq_attr *mqstat);
/* get attributes */
int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);

/* register for notification if message becomes available */
int mq_notify(mqd_t mqdes, const struct sigevent *notification);
```

Concurrency

IPC communication using signals

Signals

- **Signals** are typically used by the operating system to signal certain events to processes.
- Signals can have several reasons
  - Exception, e.g. division by zero (SIGFPE) or segmentation fault (SIGSEGV)
  - Reactions to user input (e.g. ctrl +c)
  - Signals used by other processes to communicate
  - Signaling of events from the operating system, e.g. timeout, asynchronous I/O operation finished, message arrival at an empty message queue (see `mq_notify()`)

Process signal-handling

- Processes can react to signals in 3 different ways:
  1. Ignore signal
  2. Execute signal-handler
  3. Delay the signal until process is ready for reacting
- Additionally there is the possibility to use the default-handler for signals. Very often the default treatment of signals is to abort the program, therefore program should have a reasonable signal handling if certain signals are expected to occur.

Semaphores for signaling events

- Semaphores can also be used for signaling events instead of mutual exclusion.
- It is reasonable, that processes (producer) permanently release and other processes (consumer) permanently acquire semaphores.
- It is also possible to create named semaphores, which can then be used between processes (instead of only threads).
- Relevant functions in this case are:
  - `sem_open()`: for creation and opening of named semaphores
  - `sem_unlink()`: for deleting named semaphores

Example: semaphores for signaling events

- A **worker** process waits for a job issued by a **contractor** to process and afterwards waits for the next job.

Worker*:

```
while( true ) {
    down( sem ); /* wait for next job */

    process( job );
}
```

Contractor*:

```
Job* job;
...
job = .... /* create new job */
up( sem ); /* signal new job */
...
```

** Much simplified solution, since there is always only one job available*

Semaphores for signaling events: problems

- One problem of the implementation on the previous slide is that the `job` pointer is not protected and therefore errors can happen.
- Using another semaphore can resolve this problem.
- If the amount of time between two jobs is to small, then two additional problems can occur:
  - Problem 1: the **contractor** has to wait, because the **worker** process still processes the last job.
  - Problem 2: The last job is overwritten, if it hasn't been processed yet. Depending on the semaphore implementation it can also happen, that the new job is executed twice.

⇨ Semaphores are only reasonable for simple signaling-problems (without data transfer). Otherwise message queues should be used.

Semaphores for signaling events: reader-writer example

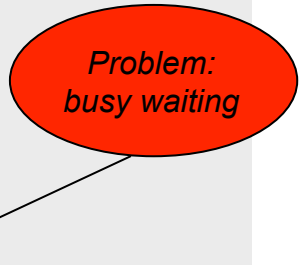
- Previous solution:

Reader

```
...
    down(semWriter);
    down(semCounter);
    rcounter++;
    up(semCounter);
    up(semWriter);

    read();

    down(semCounter);
    rcounter--;
    up(semCounter);
...
```

Writer

```
...
    down(semWriter);

    while(true)
    {
        down(semCounter);
        if(rcounter==0)
            break;
        up(semCounter);
    }
    up(semCounter);

    write();

    up(semWriter);
...
```



Semaphores for signaling events: reader-writer example

- Solution using signaling:

Reader

```
...
    down(semWriter);
    down(semCounter);
    rcounter++;
    if(rcounter==1)
        down(semReader);
    up(semCounter);
    up(semWriter);

    read();

    down(semCounter);
    rcounter--;
    if(rcounter==0)
        up(semReader);
    up(semCounter);
...
```

Writer

```
...
    down(semWriter);
    down(semReader);
    up(semReader);

    write();

    up(semWriter);
...
```

Barriers

- **Definition**: a barrier for a set of processes is a code location, that all processes need to reach before execution can be continued by any process.
- The special case of two processes is called Rendezvous (e.g. Ada programming language)
- Barriers can be implemented using semaphores